VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ

BRNO UNIVERSITY OF TECHNOLOGY

FAKULTA STROJNÍHO INŽENÝRSTVÍ

FACULTY OF MECHANICAL ENGINEERING

ÚSTAV MATEMATIKY

INSTITUTE OF MATHEMATICS

HYBRIDNÍ ALGORITMY V OPTIMALIZACI

HYBRID ALGORITHMS IN OPTIMIZATION

BAKALÁŘSKÁ PRÁCE

BACHELOR'S THESIS

AUTOR PRÁCE — **PETR ZAMAZAL**

AUTHOR

VEDOUCÍ PRÁCE — **RNDr. PAVEL POPELA, Ph.D.**

SUPERVISOR

BRNO 2018

VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
FAKULTA STROJNÍHO INŽENÝRSTVÍ

Zadání bakalářské práce

Ústav:	Ústav matematiky
Student:	**Petr Zamazal**
Studijní program:	Aplikované vědy v inženýrství
Studijní obor:	Matematické inženýrství
Vedoucí práce:	**RNDr. Pavel Popela, Ph.D.**
Akademický rok:	2017/18

Ředitel ústavu Vám v souladu se zákonem č.111/1998 o vysokých školách a se Studijním a zkušebním řádem VUT v Brně určuje následující téma bakalářské práce:

Hybridní algoritmy v optimalizaci

Stručná charakteristika problematiky úkolu:

Student se seznámí s problematikou optimalizačních úloh a jejich aplikacemi v inženýrství. Důraz bude kladen na jejich algoritmické řešení pomocí deterministických, stochastických a heuristických algoritmů. Student při zpracování práce využije osvojené poznatky jak matematické analýzy a lineární algebry, tak dále diskrétní matematiky, algebry a programování k vhodnému návrhu hybridního algoritmu pro řešení vybrané třídy optimalizačních úloh.

Cíle bakalářské práce:

1. Sestavení přehledu základních poznatků pro vybranou třídu optimalizačních úloh.
2. Uvedení vybraných klasických a heuristických algoritmů.
3. Vytvoření hybridního algoritmu pro vybranou úlohu.
4. Implementace algoritmu a jeho testování.
5. Analýza a interpretace výsledků.

Seznam doporučené literatury:

KLAPKA, Jindřich, Jiří DVOŘÁK a Pavel POPELA. Metody operačního výzkumu. Vyd. 2. Brno: VUTIUM, 2001. ISBN 80-214-1839-7.

WOLSEY, Laurence A. Integer programming. New York: John Wiley & Sons, 1998. ISBN 978-0-4-1-28366-9.

CHRISTOFIDES, Nicos. Graph Theory - an Algorithmic Approach. Academic Press, 1975. ISBN 0-1--174350-0.

REEVES, Colin R. Modern Heuristic Techniques for Combinatorial Problems. Wiley and Sons, 1993. ISBN 978-0470220795

PARDALOS, Panos M. a Mauricio G. C. RESENDE (eds.). Handbook of applied optimization. Oxford: Oxford University Press, 2002. ISBN 0195125940.

Termín odevzdání bakalářské práce je stanoven časovým plánem akademického roku 2017/18

V Brně, dne

L. S.

prof. RNDr. Josef Šlapal, CSc.
ředitel ústavu

doc. Ing. Jaroslav Katolický, Ph.D.
děkan fakulty

Abstrakt

Tato práce se zabývá řešením úlohy celočíselného programování pomocí hybridního algoritmu. Jde o úlohu hledání toku v síti s nejmenšími náklady s možností přidávání nových hran. Zmíněný hybridní algoritmus je založen na genetickém algoritmu za použití síťové simplexové metody. Implementován je v programovacím jazyce Python.

Summary

This work deals with the solution of an integer programming task using a hybrid algorithm. Mentioned task is a minimum cost network flow problem with option of adding new edges. The hybrid algorithm is based on a genetic algorithm using the network simplex method. Implementation is in the Python programming language.

Klíčová slova

Síťová úloha, hybridní algoritmus, genetický algoritmus, Python

Keywords

Network problem, hybrid algorithm, genetic algorithm, Python

ZAMAZAL, P.*Hybridní algoritmy v optimalizaci*. Brno: Vysoké učení technické v Brně, Fakulta strojního inženýrství, 2018. 33 s. Vedoucí RNDr. Pavel Popela, Ph.D.

Prohlašuji, že jsem bakalářskou práci *Hybridní algoritmy v optimalizaci* vypracoval samostatně, za odborného vedení vedoucího bakalářské práce RNDr. Pavla Popely, Ph.D. Dále prohlašuji, že veškeré podklady, ze kterých jsem čerpal jsou uvedeny v seznamu použité literatury

Petr Zamazal

Zde bych rád poděkoval zejména svému vedoucímu práce panu RNDr. Pavlu Popelovi, Ph.D. za konzultace, odborné vedení, trpělivost a podněty k práci. Dík také patří mé rodině a přátelům za jejich podporu.

Petr Zamazal

Obsah

1. Úvod

Existují problémy u nichž je časová náročnost řešení s dnešními výpočetními možnostmi příliš vysoká. Tím vzniká motivace pro tvorbu hybridních algoritmů využívajících heuristik. Na úkor přesnosti a předvídatelnosti jsou heuristické algoritmy schopny relativně rychlého řešení.
Toto práce si dává za cíl vytvořit hybridní algoritmus pro úlohu celočíselného programování. Konkrétně jde o genetický algoritmus používající síťovou simplexovou metodu a problém hledání toku sítí s minimálními náklady při možnosti přidávání nových hran.
Druhá kapitola obsahuje základní pojmy lineárního programování, pomocí nichž následně popisuje způsob řešení úloh lineárního programování, simplexovou metodu. Na konci kapitoly je stručný popis hybridních, deterministických a heuristických algoritmů.
Další kapitola se zabývá konkrétní heuristikou, a to genetickým algoritmy. Popisuje části, z kterých se genetický algoritmus skládá a jejich různé možné varianty.
Čtvrtá kapitola nejdříve definuje několik pojmů z teorie grafů, pomocí nichž potom popisuje úlohu toku sítí při minimálních nákladech. Jde o úlohu lineárního programování, na které je založen problém řešený v této práci.
Ten je popsaný na začátku následující kapitoly. Ta potom pokračuje postupným popisem částí genetického algoritmu aplikovaného na síťovou úlohu s možností přidávání hran. Zároveň je popsán zdrojový kód programu realizujícího zmíněný algoritmus. Zvoleným programovacím jazykem je jazyk Python. Na konci je popsána testovací úloha s konkrétními parametry a výstupy z na ní testovaného programu.

2. Optimalizace

2.1. Lineární programování

Definice 2.1. *Nechť a_{ij}, b_i, c_j jsou daná reálná čísla, $\mathbf{x} = (x_j), \mathbf{b} = (b_i), \mathbf{c} = (c_j)$ jsou vektory a $\mathbf{A} = (a_{ij})$ je matice typu $m \times n$, kde $i = 1, ..., m; j = 1, ..., n$. Dále nechť x_j jsou reálné proměnné. Poté úlohu minimalizace funkce*

$$\mathbf{c}^\top \mathbf{x},$$

za podmínek

$$\mathbf{A}\mathbf{x} = \mathbf{b},$$

$$\mathbf{x} \geq \mathbf{0},$$

nazveme minimalizační úlohou lineárního programování ve standardním tvaru.

Standardní tvar je vhodný pro většinu algoritmů.

Poznámka 2.1. *Pokud je část (jejich počet označím q) podmínek ve tvaru nerovnic, převedeme je do tvaru*

$$\sum_{j=1}^{n} a_{kj} x_j \leq b_k$$

kde $k = 1, ..., q$. a následně vytvoříme vektor dalších proměnných $\mathbf{s} = (s_k)$ tak, aby platilo

$$\sum_{j=1}^{n} a_{kj} x_j + s_k = b_k, \qquad \mathbf{s} \geq \mathbf{0}.$$

Definice 2.2. *Množinu vektorů $\mathbf{x}$ splňující $\mathbf{A}\mathbf{x} \leq \mathbf{b}$ a $\mathbf{x} \geq \mathbf{0}$ nazvu množinou přípustných řešení dané úlohy.*

Definice 2.3. *Řešení $\mathbf{x}_o$ z množiny přípustných řešení nazveme optimálním řešením dané úlohy, pokud*

$$\mathbf{c}^\top \mathbf{x}_o \geq \mathbf{c}^\top \mathbf{x},$$

kde $\mathbf{x}$ jsou vektory z množiny přípustných řešení.

Definice 2.4. *Množinu $K \subset R^n$ nazveme konvexní množinou, pokud pro libovolné dva body $\mathbf{x}, \mathbf{y}$ z množiny K a libovolné $\alpha \in (0, 1)$ platí*

$$\alpha\mathbf{x} + (1 - \alpha)\mathbf{y} \in K.$$

Definice 2.5. *Konvexní polyedrickou množinou $M \subset R^n$ nazveme takovou množinu, kterou lze vyjádřit jako průnik konečného počtu uzavřených poloprostorů. Hranice těchto poloprostorů nazveme vytvářejícími nadrovinami množiny M.*

Věta 2.1. *Množina přípustných řešení úlohy lineárního programování je konvexní polyedrickou množinou.*

Definice 2.6. *Nechť $K \subset R^n$ je libovolná množina. Bod $\mathbf{k} \in K$ nazveme krajním bodem množiny, pokud neexistují dva body $\mathbf{x}, \mathbf{y} \in K$ a číslo $\alpha \in (0, 1)$ takové, že $\mathbf{x} \neq \mathbf{y}$ a $\mathbf{k} = \alpha\mathbf{x} + (1 - \alpha)\mathbf{y}$.*

Věta 2.2. *Konvexní polyedrická množina má konečný počet krajních bodů.*

Definice 2.7. *Nechť $K \subset R^n$ je konvexní polyedrická množina a L je její neprázdná podmnožina. Pokud lze L vyjádřit jako průnik množiny K a příslušných jejích vytvářejících nadrovin, které obsahují L, nazveme L stěnou množiny K.*

Definice 2.8. *Jednorozměrnou stěnu nazveme hranou.*

Doplňující informace a vlastnosti je možná hledat v [6] a [8].

2.1.1. Simplexová metoda

Nejznámějším algoritmem pro řešení úloh lineárního programování je simplexová metoda. Existují ale i jiné, asymptoticky rychlejší algoritmy, například metoda vnitřních bodů. Ve své práci používám Simplexovou metodu, protože ji znám z výuky a je vhodná pro daný typ úloh. Metoda byla odvozena Georgem Dantzigem v roce 1947. Její princip se dá dobře vysvětlit geometricky. Předpokládáme, že známe krajní bod $\mathbf{x_1}$ z množiny přípustných řešení. Z tohoto bodu vychází konečný počet hran, které buď obsahují další krajní bod, nebo jsou neomezené. Pokud některá z neomezených hran obsahuje bod s větší hodnotou účelové funkce než $\mathbf{c}^\top\mathbf{x}_1$, pak úloha nemá optimální řešení. V opačném případě hledáme hranu jejíž další krajní bod má hodnotu účelové funkce vyšší než $\mathbf{c}^\top\mathbf{x}_1$, takový bod potom označíme $\mathbf{x}_2$ a provádíme od začátku stejný postup znovu. Pokud žádný takový krajní bod neexistuje, nalezli jsme optimum a krajní bod, ve kterém se nalézáme je optimálním řešením. Pokud v tu chvíli existují hrany, jejichž další krajní body mají stejnou hodnotu účelové funkce, tak jsou řešením také tyto krajní body i body na těchto hranách. Konečný počet krajních bodů konvexní polyedrické množiny zaručuje, že algoritmus po konečném počtu kroků nalezne optimum, nebo zjistí, že úloha nemá optimální řešení. Podrobněji je algoritmus popsaný v [1], [6] a [8].

2.2. Hybridní algoritmus

Hybridní algoritmus se skládá z deterministického a heuristického algoritmu, kdy jeden začíná tam, kde druhý přestal, nebo se doplňují. Hybridní algoritmy vznikají ve snaze využít univerzálnosti heuristik a přesnosti klasických algoritmů.

2.2.1. Deterministický algoritmus

Deterministickým nazveme algoritmus, který při stejných vstupech vytvoří stejné výstupy za stejný čas. Takový algoritmus je v podstatě matematickou funkcí, která s danými vstupy vytvoří jednoznačný výstup. Příkladem deterministického algoritmu je uvedená simplexová metoda.
Problémem deterministických algoritmů je, že pro některé problémy nemusí existovat, nebo jsou velmi pomalé.

2.2.2. Heuristický algoritmus

Heuristický algoritmus využívá náhodnosti ke zkusmému řešení problému za vhodných podmínek, které zvyšují pravděpodobnost nalezení co nejlepšího výsledku. Používá se

ve chvíli kdy neexistuje vhodný deterministický algoritmus, nebo je příliš pomalý. Kvůli náhodnosti nemůže heuristika zaručit dostatečně přesné řešení v určitém čase.
Při tvorbě heuristických algoritmů se často jako inspirace používá příroda a vzorce chování v ní.

Náhodné prohledávání

Jde o jeden z nejjednodušších heuristických algoritmů. Lze na něm dobře pozorovat jejich vlastnosti. Jeho princip spočívá v náhodném generování bodů z množiny přípustných řešení a uchovává nejlepší dosavadní zaznamenaný výsledek.

3. Genetické algoritmy

Genetické algoritmy jsou heuristikou, která je inspirovaná principem přirozeného výběru. Spočívá ve vytvoření počáteční populace, kde jedinci představují jednotlivá řešení. Následně algoritmus tvoří nové generace vždy na základě té předešlé pomocí genetických operátorů tak, aby byly nové generace lepší než předešlé. Další informace o genetických algoritmech viz [5], [7] a [10].

3.1. Podoba jedinců

Aby fungovaly principy následujících genetických operátorů, je třeba aby jedinci (jednotlivá řešení) byli zapsáni ve formě, která se skládá ze sekvencí a změna jedné sekvence příliš nezměnila samotného jedince. Tuto formu nazveme chromozomem a sekvence nazveme geny. Nejčastěji se používá zápis binárním chromozomem, tedy konečnou posloupností jedniček a nul, do kterého je třeba vhodně jedince zakódovat. Například kódování celých čísel pomocí obvyklého binárního zápisu je nevhodné, protože rozdíl čísel 2^n a $2^n - 1$ je 1, ale jejich binární kód se liší v $n + 1$ pozicích. Pro kódování celých čísel se proto používá tzv. Grayův kód.

3.2. Ohodnocení (fitness)

Pro použití dalších operátorů a pro ověření nalezení výsledku je třeba každého jedince ohodnotit. K tomu se používá fitness funkce, která určuje hodnotu tzv. fitness. V optimalizaci může fitness funkce odpovídat účelové funkci. V takovém případě při minimalizaci považujeme jedince s menším fitness za lepšího, nebo provedeme vhodnou transformaci. Fitness funkce musí nabývat svého maxima (minima při minimalizaci) v místě hledaného řešení.

3.3. Genetické operátory

3.3.1. Selekce

Operátor selekce slouží k výběru jedinců pro další operátory. Selekce by měla splňovat, že pravděpodobněji vybírá jedince s lepším fitness, a výběr by měl být podmíněn náhodou. První požadavek je z důvodu zlepšování populace a druhý je kvůli zachování rozmanitosti populace.
Základní typy selekce jsou:

Ruletový systém

Pravděpodobnost výběru jedince závisí na jeho fitness hodnotě. Název je odvozen ze zobrazení na ruletovém kole, které je rozděleno na výseče, kde každá výseč odpovídá jednomu jedinci z dané generace. Poměr velikostí výsečí vyplývá z poměru fitness jedinců, nebo její transformace. Samotný výběr probíhá vygenerováním náhodného úhlu s rovnoměrným rozdělením pravděpodobnosti, který určí jedince podle výseče, ve které leží.

Podle pořadí

V této metodě nezáleží na konkrétní hodnotě fitness, ale na pořadí těmito hodnotami určeném. Pravděpodobnost výběru je dána funkcí závislou na pořadí jedince. Podle zvolené pravděpodobnostní funkce se určuje tzv. selektivní tlak, který je zaveden poměrem pravděpodobnosti volby nejlepšího jedince a průměrného jedince.

3.3.2. Křížení

Křížení je genetický operátor, který na základě dvou rodičovských jedinců vybraných selekcí vygeneruje jedince nového.
Běžnými typy křížení jsou:

Jednobodové křížení

Mějme dva jedince s chromozomy $(a_1, a_2, ..., a_n)$ a $(b_1, b_2, ..., b_n)$ délky n. Náhodně zvolíme číslo i, tak aby $1 \leq i < n$. Nový jedinec bude mít potom chromozom $(a_1, ..., a_i, b_{i+1}, ..., b_n)$

Vícebodové křížení

Od jednobodového křížení se liší tím, že náhodně zvolených čísel je více a rodičovské kódy se tím pádem střídají vícekrát.

Uniformní křížení

V tomto případě je u každého genu náhodně zvoleno, ze kterého rodičovského chromozomu bude pocházet.

3.3.3. Mutace

Tento operátor mírně obměňuje chromozom jedince za účelem vzniku nových genů v populaci. Heuristické algoritmy jsou náchylné k uváznutí v lokálním minimu, proto je potřeba mutace, která umožňuje toto lokální minimum opustit.
Možné podoby mutace:

Jednobodová mutace

Každý jedinec má danou pravděpodobnost, že jeden jeho náhodně vybraný gen zmutuje. V případě binárního chromozomu se nula změní na jedničku a naopak. Tento typ mutace je celkem slabý a ovlivňuje průběh algoritmu relativně málo.

Uniformní mutace

Každý gen každého jedince má stejnou pravděpodobnost, že zmutuje.

3.4. Princip obměny populace

Ze staré generace se pomocí genetických operátorů vytvoří skupina potomků a je třeba určit složení nové generace.
Ryzí obměna je přístup, kdy je nová generace složena pouze z potomků. Dochází zde k riziku, že budou zahozeni dobře hodnocení jedinci. Pokud z předchozí generace zachováme určitý počet nejlepších jedinců nazveme postup **elitním doplněním**. Další možností je **elitní doplnění podle ohodnocení**, kdy je vybráno určité množství nejlepších potomků a určité množství nejlepších jedinců z předchozí generace. Aby počet seděl, je odebráno takové množství nejhorších potomků, aby měla nová generace stejný počet jedinců jako stará.

3.5. Podmínka pro ukončení algoritmu

Je několik způsobů jak definovat podmínku pro ukončení algoritmu.

1. Dosažení předem určeného maximálního počtu iterací
2. Předem daný počet generací po sobě je nejlepší jedinec stejný
3. Přiblížení se k hledanému řešení s danou přesností (vyžaduje znalost hledaného řešení, tato varianta je tedy vhodná pro testovací úlohy)

4. Síťová úloha

4.1. Teorie grafů

Abych mohl popsat síťovou úlohu musím zavést některé základní pojmy teorie grafů. Informace o teorii grafů jsem čerpal z [2].

Definice 4.1. *Grafem nazveme uspořádanou dvojici $G = (V, E)$, kde V je konečná množina vrcholů a E je množinou dvouprvkových podmnožin množiny V navzájem různých vrcholů. E nazveme množinou hran. Pro značení množiny vrcholů V, nebo množiny hran E grafu G se používá zápisu $V(G)$, respektive $E(G)$.*

Definice 4.2. *Orientovaným grafem nazveme uspořádanou dvojici $G = (V, E)$, kde V je konečná množina vrcholů a E je podmnožinou kartézského součinu $V \times V$. E nazveme množinou orientovaných hran.*

Definice 4.3. *Graf H nazveme podgrafem grafu G pokud splňuje $V(H) \subseteq V(G), E(H) \subseteq E(G)$. Analogicky definujeme orientovaný podgraf.*

4.2. Formulace síťové úlohy

Základní síťový optimalizační problém je hledání toku v síti s co nejmenšími dopravními náklady. Sítí je zde myšlen orientovaný graf, kde každý vrchol má přiřazené množství zboží, které buď produkuje nebo spotřebovává. Kladná hodnota značí produkci, záporná spotřebu a nulová značí vrchol, přes který je zboží pouze přepravováno. Budeme předpokládat, že výroba se rovná spotřebě a součet hodnot všech vrcholů je tedy nulový. Hrany tohoto grafu mají přiřazenou cenu za přepravu jednotkového množství zboží po této hraně. Cílem je najít orientovaný podgraf představující tok v síti s nejmenší cenou za přepravu zboží ze zdrojů do místa spotřeby.
Tato úloha je úlohou lineárního programování, můžeme ji tedy zapsat ve tvaru minimalizace

$$\mathbf{c}^\top \mathbf{x},$$

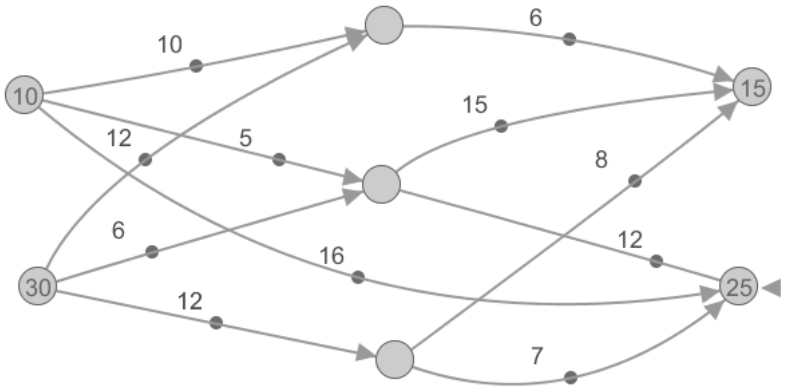
za podmínek

$$\mathbf{A}\mathbf{x} = \mathbf{b},$$

$$\mathbf{x} \geq \mathbf{0},$$

kde $\mathbf{c} = (c_j)$, c_j je cena za přepravu jednotky zboží po hraně j ($j = 1, ..., n$, n je počet hran). $\mathbf{b} = (b_i)$, b_i je spotřeba, či produkce ve vrcholu i ($i = 1, ..., m$, m je počet vrcholů). $\mathbf{x} = (x_j)$ x_j je množství přepravovaného zboží po hraně j. $\mathbf{A} = (a_{ij})$ je tzv. matice incidence, která popisuje, ze kterého a do kterého vrcholu daná hrana vede. a_{ij} je rovno 0, pokud j-tá hrana nezačíná ani nekončí ve vrcholu i, pokud j-tá hrana začíná ve i-tém vrcholu, tak je rovno 1, a pokud v něm hrana končí, tak je a_{ij} rovno -1. Tím pádem $\mathbf{A}\mathbf{x} = \mathbf{b}$ tvoří soustavu rovnic, kde každá rovnice popisuje bilanci pro určitý vrchol. Na levé straně je záporná hodnota zboží z hran končících ve vrcholu a kladné množství zboží odtékající hranami ve vrcholu začínajícími. Na pravé je případná výroba nebo spotřeba v daném vrcholu.

K řešení síťových úloh se používá síťová simplexová metoda, která je modifikovanou simplexovou metodou tak, aby využívala řídkosti incidenční matice a dalších specifik síťových úloh. Bližší informace k síťovým úlohám jsou v [1] a [8] a síťová simplexová metoda je rozvedena v [8].
Problém ilustruji příkladem popsaném obrázkem 4.1. Úlohu zapíši následovně:



Obrázek 4.1: Graf sítě s uvedenými cenami přepravy a množstvím výroby a spotřeby.

Minimalizuj z:

$$z = 10x_1 + 5x_2 + 16x_3 + 12x_4 + 6x_5 + 12x_7 + 6x_1 + 15x_8 + 12x_9 + 8x_{10} + 12x_{11}$$

za podmínek

$$\begin{aligned}
x_1 + x_2 + x_3 &= 10 \\
x_4 + x_5 + x_6 &= 30 \\
x_7 - x_1 - x_4 &= 0 \\
x_8 + x_9 - x_2 - x_5 &= 0 \\
x_{10} + x_{11} - x_6 &= 0 \\
-x_7 - x_8 - x_{10} &= -15 \\
-x_3 - x_9 - x_{11} &= -25 \\
0 \leq x_1, x_2, x_3, x_4, x_5, x_6, x_7, x_8, x_9, x_{10}, x_{11}
\end{aligned}$$

Maticově vektorové vyjádření se potom skládá z vektoru cen

$$\mathbf{c} = (10\ 5\ 16\ 12\ 6\ 12\ 6\ 15\ 12\ 8\ 12),$$

vektoru pravých stran podmínek, tedy hodnot výroby a spotřeby

$$\mathbf{b} = (10\ \ 30\ \ 0\ \ 0\ \ 0\ \ -15\ \ -25),$$

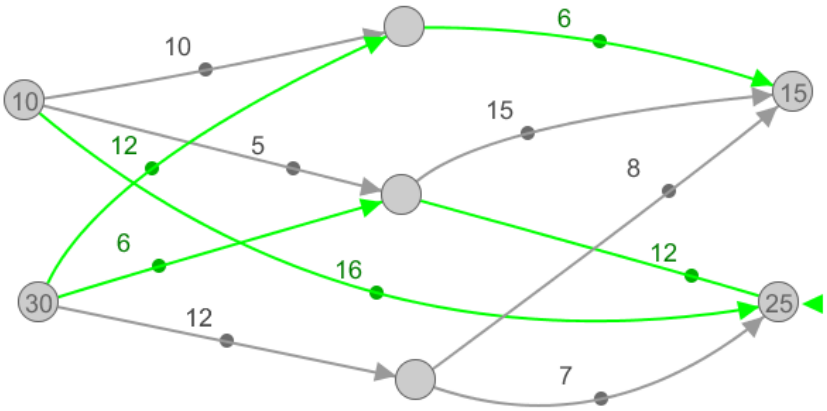
a matice incidence

$$\mathbf{A} = \begin{pmatrix} 1 & 1 & 1 & & & & & & & & \\ & & & 1 & 1 & 1 & & & & & \\ -1 & & & -1 & & & 1 & & & & \\ & -1 & & & -1 & & & 1 & 1 & & \\ & & & & & -1 & & & & 1 & 1 \\ & & & & & & -1 & -1 & & -1 & \\ & & -1 & & & & & & -1 & & -1 \end{pmatrix}$$

Řešením této úlohy je

$$\mathbf{x} = (0\ 0\ 10\ 15\ 15\ 0\ 15\ 0\ 15\ 0\ 0),$$

v obrázku 4.2 je vyznačené zeleně.



Obrázek 4.2: Graf sítě s vyznačenými hranami výsledného toku.

5. Řešený problém

5.1. Popis úlohy

Podklady pro tuto úlohu jsem čerpal z [4]. Úloha na jejíž řešení se v této práci zaměřuji, je modifikací síťové úlohy popsané v 4.2. V této variantě je možné zapínat v síti za poplatek nové hrany. Jako aplikaci na reálný problém je možné si představit jednorázový poplatek při využití nějaké trasy, který nezávisí na přepraveném množství. V malém měřítku to může být dálniční známka, ve velkém měřítku to může být platba za pronájem železnice, či náklady na opravu silnice.
Na začátek budu uvažovat, že nemám zapnutou žádnou hranu a formulace síťové úlohy v 4.2 mi popisuje hrany, které mohu zapnout. Úlohu je potom třeba doplnit vektorem proměnných $\mathbf{y} = (y_j)$, kde j je číslo hrany a $y_j \in \{0, 1\}$ je proměnná, která určuje, jestli je hrana zapnutá, nebo ne (pokud $y_j = 1$, tak je j-tá hrana zapnutá a pokud $y_j = 0$, tak je vypnutá), a vektorem $\mathbf{q} = (q_j)$ cen za zapínání hran. Potom úlohu můžeme formulovat jako minimalizaci funkce

$$\mathbf{c}^\top \mathbf{x} + \mathbf{q}^\top \mathbf{y},$$

za podmínek

$$\mathbf{A}'\mathbf{x} = \mathbf{b},$$

$$\mathbf{0} \leq \mathbf{x} \leq u\mathbf{y},$$

kde $\mathbf{A}'$ je matice vytvořená z matice incidence $\mathbf{A}$ vynulováním sloupců odpovídajícím vypnutým hranám (nulám na příslušných pozicích vektoru $\mathbf{y}$). u je mezní tok hranou, to znamená větší z čísel součet celkové výroby a součet celkové spotřeby. Úloha ve které je už na začátku zapnutá část hran se liší tím, že některé prvky vektoru $\mathbf{y}$ zafixujeme na hodnotě 1 a příslušné prvky $\mathbf{q}$ změním na 0.
Tato úloha pracuje se dvěma soubory neznámých, z nichž jeden je celočíselný. Tím pádem jde o úlohu, kterou je třeba řešit metodami celočíselného programování nebo jako v případě této práce, heuristikou.
Ideou řešení bude rozdělit úlohu na dvě úrovně. V základní úrovni budeme mít klasickou síťovou úlohu pro aktuálně zapnuté hrany. Další úrovní bude genetický algoritmus, který řešení jednotlivých síťových úloh použije jako fitness 3.2 jedinců tvořených příslušnými vektory $\mathbf{q}$.

5.2. Python

Algoritmus pro řešení úlohy jsem se rozhodl vytvářet v programovacím jazyce Pythonu jakožto jednom z nejpoužívanějších programovacích jazyků dnešní doby, protože je volně dostupný, existuje k němu rozsáhlé množství knihoven a nástrojů a chtěl jsem rozšířit svoji znalost tohoto programovacího jazyka. Řešič pro síťovou simplexovou metodu používám z balíčku NetworkX a pro práci s maticemi balíček NumPy. Dokumentaci k Pythonu lze nalézt v [9]

5.3. Popis programu

5.3.1. Vstupy, nahrávání dat a jejich předzpracování

Základní vstupní data, která program potřebuje jsou informace o konkrétním příkladu viz 5.1 a 4, tedy vektor cen za zapnutí hran **q**, v programu značený *cena_stavby*, a vektor **b** výroby a spotřeby, v programu také značený *b*, ve kterém jsou záporná znaménka u výroby a kladná u spotřeby (tuto formu vyžaduje příkaz volající síťovou simplexovou metodu). V programu nepracuji s maticí incidence, proto místo toho nahrávám údaje o počátečních a koncových bodech hran a k nim připojím vektor cen za přepravu hranou **c**. Výslednou třířádkovou matici nazývám v programu jako *trasy*. Program nahrává data z přiloženého souboru MS Excel za pomoci knihovny xlrd.
Na začátku ze vstupních dat určím pomocné parametry vektoru počáteční trasy *poc_trasa*, obsahujícího hodnoty 1 pro zapnutou hranu a 0 pro vypnutou hranu, a vektoru hran *indexy_moznych_tras*, které je možné zapnout.
Parametry algoritmu jako podmínky pro ukončení, počet jedinců v generaci a další je třeba měnit přímo v programu. Nemělo by být těžké upravit kód a v případě potřeby je také nahrávat z externího zdroje.
Kód funkce pro import dat:

```
def import_dat():
  workbook = xlrd.open_workbook('data.xls')      # otevreni souboru MS Excel
  worksheet = workbook.sheet_by_name('data1')    # otevreni prislusneho
                                                 listu

  # import cen zapinani hran
  i=1
  cena_stavby=[]
  while worksheet.cell(i, 3).value != 'stop':
    cena_stavby=np.append(cena_stavby, [worksheet.cell(i, 3).value], axis=0
                                                 )
    i=i+1
  cena_stavby1=cena_stavby.tolist()

  # import pocatecnich a koncovych bodu hran a cen prepravy
  trasa=np.zeros((3, len(cena_stavby1)))
  i=1
  while worksheet.cell(i, 0).value != 'stop':
    trasa[0,i-1]=worksheet.cell(i, 0).value
    trasa[1,i-1]=worksheet.cell(i, 1).value
    trasa[2,i-1]=worksheet.cell(i, 2).value
    i=i+1

  # import vyroby a spotreby
  i=1
  b=[]
  while worksheet.cell(i, 4).value != 'stop':
    b=np.append(b, [worksheet.cell(i, 4).value], axis=0)
    i=i+1
  b=b.astype(int)
  b=b.tolist()
  return (cena_stavby1,trasa.astype(int), b)
```

5.3.2. Popis jedinců a jejich ohodnocení

Jedinci v mém algoritmu jsou vektory délky $j + 1$, kde jsou na prvních j místech hodnoty 1 pro zapnuté hrany a 0 pro vypnuté hrany. Na poslední pozici je fitness jedince, které určíme jako řešení síťové úhly pro takto zapnuté hrany síťovou simplexovou metodou s přičtenými náklady za zapnutí hran pro tvorbu daného jedince. Jedince s nižším fitness považuji proto za lepší. Původně jsem v programu používal obyčejnou simplexovou metodu. Po implementaci síťové simplexové metody došlo při testování na stejných příkladech ke snížení časové náročnosti přibližně dvacetinásobně. Počáteční zapnutou síť považuji za nevypnutelnou, proto jakákoli úprava, či generování nemůže způsobit nulovou hodnotu pro hranu, co byla na začátku zapnutá. Předpokládám, že počáteční síť má řešení, jinak by pro některé jedince neexistovalo fitness ohodnocení. Pokud zadaná síť není řešitelná, je třeba ručně přidat umělé hrany, tak aby úloha měla řešení. Těmto umělým hranám přiřadím velmi vysokou cenu přepravy tak, aby se program snažil co nejdříve vytvořit jinou trasu. Je možné, aby mezi dvěma body vedlo více různě ohodnocených hran. Populaci chápu jako matici, jejíž řádky jsou tvořeny jedinci.
Funkce pro ohodnocení populace:

```
def fitness(populace, trasa, b, cena_stavby):
  for i in range(populace.shape[0]):
    if populace[i,-1]==0:
      # priprava dat pro prikaz volajici sitovou simplexovou metodu
      k=(np.array(populace[i,0:-1]) * np.array(trasa[0,:])).astype(int)
      G = nx.DiGraph()
      for j in range(1,(len(b)+1)):
        G.add_node((j),demand=b[j-1])
      for j in range(trasa.shape[1]):
        if k[j]!=0:
          G.add_edge(k[j], trasa[1,j],weight=trasa[2,j])
      # sitova simplexova metoda
      res=nx.network_simplex(G)
      # prirazeni fitness s pripoctenymi poplatky za zapinani hran
      price=res[0]+sum(cena_stavby*populace[i,0:-1].T)
      populace[i,-1]=price
  # serazeni populace podle fitness
  populace1=populace[np.argsort(populace[:, -1])]
  return populace1.astype(int)
```

5.3.3. Genetické operátory

Selekci v mém algoritmu provádím ruletovým systémem (viz. 3.3.1) tak, že transformuji fitness k-tého jedince v populaci (dále budu značit jako $f(k)$)

$$F(k) = \frac{f_{max} - f(k)}{f_{max} - f_{min}} + \frac{1}{N},$$

kde f_{max} je nejvyšší dosažené fitness v populaci a f_{min} je nejnižší. N je počet prvků populace. Samotnou pravděpodobnost vybrání jedince dostanu jako

$$P(k) = \frac{F(k)}{\sum_{i=1}^{N} F(i)}.$$

Křížení provádím jednobodově (viz 3.3.2) ze dvou jedinců vybraných selekcí.
Mutaci potom provádím uniformně (viz. 3.3.3) na jedincích nově vzniklých křížením.
Kromě uvedených genetických operátorů používám mechanizmus odvozený z vlastností řešeného problému, kdy u selekcí zvoleného jedince vypnu hrany (ne počáteční), kterými v optimalizovaném řešení neprochází žádný tok. Tento nový operátor nazvu redukcí. Pomáhá očistit jedince od zbytečně zapnutých hran a snižuje tak jejich fitness. Negativním efektem je ubírání genů z populace, protože více různých jedinců má stejný optimální tok. Na takto nově vzniklé jedince již neaplikuji mutaci.
Kód pro tvorbu potomků:

```
def potomci(populace,n_potomku,n_krizencu, mutace, trasa, cena_stavby,
                                        indexy_moznych_tras, b, poc_trasa):
  velikost_populace=populace.shape[0]
  # priprava pravdepodobnostni funkce pro selekci
  spektrum=np.zeros(velikost_populace)
  for i in range(velikost_populace):
    spektrum[i]=spektrum[i-1]+((populace[velikost_populace-1,-1]-populace[i
                                        ,-1])/(populace[velikost_populace-1,-
                                        1]-populace[0,-1]))+1/
                                        velikost_populace
  spektrum=spektrum/spektrum[-1]
  potomci=np.zeros((n_potomku,populace.shape[1]))
  for i in range(n_krizencu):
    # selekce rodicovskych paru
    counta=0
    aa=random.random()
    while(aa>spektrum[counta]):
      counta=counta+1
    countb=0
    bb=random.random()
      while(bb>spektrum[countb]):
    countb=countb+1
    # krizeni
    cut=random.choice(range(populace.shape[1]-1))
    potomci[i,0:cut]=populace[counta,0:cut]
    potomci[i,cut:populace.shape[1]-1]=populace[countb,cut:populace.shape[1
                                        ]-1]

    for j in indexy_moznych_tras:
      mut=random.random()
      if mut<mutace:
        if potomci[i,j]==0:
          potomci[i,j]=1
        else:
          potomci[i,j]=0


  for i in range(n_krizencu,n_potomku):
    # selekce pro operator redukce
    counta=0
    aa=random.random()
    while(aa>spektrum[counta]):
      counta=counta+1
```

```
    potomci[i,0:-1]=populace[counta,0:-1]
    k=(np.array(potomci[i,0:-1]) * np.array(trasa[0,:])).astype(int)
    G = nx.DiGraph()
    for j in range(1,(len(b)+1)):
      G.add_node((j),demand=b[j-1])
    for j in range(trasa.shape[1]):
      if k[j]!=0:
        G.add_edge(k[j], trasa[1,j],weight=trasa[2,j])
    # ohodnoceni jedincu vybranych selekci a zjisteni toku v jejich optimu
    res=nx.network_simplex(G)
    l=np.zeros(trasa.shape[1])
    for j in range(trasa.shape[1]):
      if trasa[0,j]!=0:
        if res[1].get(trasa[0,j],0).get(trasa[1,j],0)!=0:
          l[j]=1
    # redukce
    for j in indexy_moznych_tras:
      if l[j]==0 and poc_trasa[j]==0:
        potomci[i,j]=0
    potomci[i,-1]=res[0]+sum(cena_stavby*potomci[i,0:-1].T)

  return potomci.astype(int)
```

5.3.4. Tvorba generací

První generaci vygeneruji náhodně pouze za podmínky, že nesmí být vypnuty počáteční hrany.
Další generace vytvářím z předchozí generace a jejích potomků. Nejdříve jsem používal postup, ve kterém jsem sloučil původní generaci s jejími potomky a z výsledné skupiny vybral nejlepší jedince do nové generace. Tento postup zdánlivě vede k vytváření co nejlepší populace, ale je náchylný k uváznutí v lokálním extrému, proto jsem začal používat elitní doplnění (viz. 3.4), které zachová pouze určitý počet nejlepších prvků předchozí generace a doplní ji nejlepšími potomky. Původní metodou dostatečně dobrá stará populace přešla skoro celá do generace nové, ale při elitním doplnění je umožněno vstupu novým prvkům, které jsou sice horší, ale obsahují geny, které stará generace neobsahovala.
Funkce starající se o tvorbu nových generací:

```
def nova_generace(stara_generace, potomci, trasa, b, cena_stavby, elita):
  # slouceni urciteho poctu nejlepsich jedincu z minule generace daneho
                                      vstupem "elita" s potomky
  newgen=np.concatenate((stara_generace[0:elita,:], potomci))
  # ohodnoceni a serazeni sloucene populace
  newgen=fitness(newgen, trasa, b, cena_stavby)
  # redukce poctu, aby odpovidal velikosti predchozi generace
  newgen.resize((stara_generace.shape))
  return newgen
```

5.3.5. Popis genetického algoritmu a podmínky ukončení

Schématicky lze popsat použitý genetický algoritmus posloupností kroků:

1. **Vytvoření počáteční populace** - náhodné
2. **Ohodnocení populace** - pomocí síťové simplexové metody
3. **Tvorba potomků**
   (a) **Selekce** - výběr rodičovských párů
   (b) **Křížení** - vytvoření části potomků
   (c) **Mutace** - aplikována na vzniklé potomky
   (d) **Selekce** - výběr původních jedinců
   (e) **Redukce** - jejich transformace na zbytek potomků
4. **Ohodnocení potomků** - dopočet fitness pro nové jedince
5. **Vytvoření nové generace** - elitním výběrem
6. **Kontrola koncových podmínek** - při nesplnění návrat ke kroku 3.
7. **Ukončení algoritmu** - výsledkem je nejlepší jedinec v poslední generaci

Algoritmus používá všechny tři podmínky uvedené v 3.5. Když je třeba zaměřit se na jednu z nich, ostatní nastavím nesplnitelně. Při testování algoritmu ho spustím několikrát, pouze s podmínkou maxima iterací, abych nalezl optimální řešení (přesněji nejlepší nalezené řešení, protože heuristiky ani při mnohonásobném spuštění nezaručují nalezení optima), které následně používám pro podmínku přiblížení se k optimu s danou přesností. Při provozním použití volím podmínku maximálního počtu stejných iterací nebo maxima iterací.
Zde uvedu hlavní funkci programu, která je vlastně samotným genetickým algoritmem a volá ostatní funkce. Kromě dříve popsaných vstupů daných úlohou jsou vstupem pro tuto funkci také parametry genetického algoritmu.

n_populace je počet prvků jedné generace,
n_potomků je počet generovaných potomků,
n_krizencu značí počet jedinců vzniklých křížením (zbytek potomků je doplněn redukcí),
mutace popisuje pravděpodobnost každého genu, že při mutaci změní hodnotu,
elita určuje množství nejlepších jedinců staré generace, kteří přechází do generace nové,
max_iter je maximální možný počet dosažených iterací,
konvergence určuje kolik generací po sobě může mít stejnou nejlepší hodnotu fitness,
stop je fitness, které když některý jedinec překoná, tak je ukončen algoritmus.

```
def genetic(n_populace, n_potomku, n_krizencu, mutace, trasa, cena_stavby,
                                        indexy_moznych_tras, b, poc_trasa,
                                        elita, max_iter, konvergence):
  # prvni generace a jeji ohodnoceni
  populace=prvni_generace(n_populace, poc_trasa)
  populace=fitness(populace, trasa, b, cena_stavby)
  iterace=0
  j=0
  results=[0]
```

```
  # cyklus genetickeho algoritmu
  while ((results[iterace]>10 or results[iterace]==0) and iterace<max_iter
                                    and j<konvergence ):
    # volani funkci pro tvorbu potomku a novych generaci
    potomci1=potomci(populace, n_potomku, n_krizencu, mutace, trasa,
                                    cena_stavby, indexy_moznych_tras, b,
                                    poc_trasa)
    populace=nova_generace(populace, potomci1, trasa, b, cena_stavby, elita
                                    )
    # ukladani nejlepsiho prvku populace
    results=np.append(results, [populace[0,-1]], axis=0)
    iterace=iterace+1
    if results[iterace]==results[iterace-1]:
      j=j+1
return (populace, results, iterace)
```

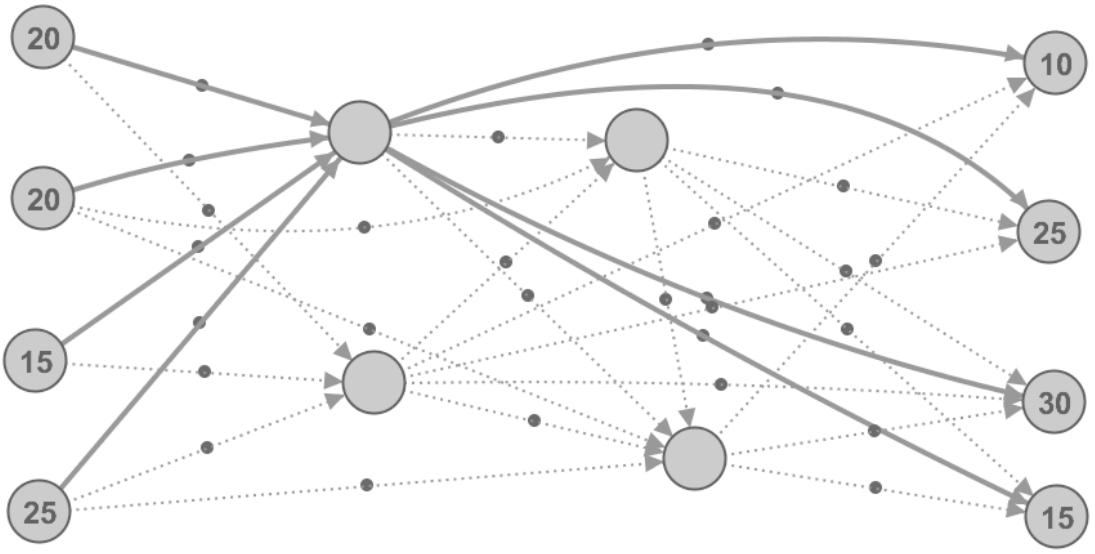
5.3.6. Výstupy a export dat

Výstupem použitého genetického algoritmu je poslední generace, nejlepší jedinec z každé generace a počet generací (iterací algoritmu). Program umožňuje zápis vektoru dat do souboru MS Excel. Další zpracování dat provádím v jádře programu nebo s exportovanými daty v prostředí MS Excel.
Funkce pro export vektoru zvolených dat do stejného souboru, z kterého importuji data pro úlohu.

```
def export_dat(export):
  export=export.tolist()
  book = xlrd.open_workbook('data.xls')
  sheet2= book.sheet_by_name('data2')
  i=0
  while sheet2.cell(0, i).value != 'stop':
    i=i+1
  book1=xlutils.copy.copy(book)
  sheet1=book1.get_sheet(1)
  for j in range(len(export)):
    sheet1.write(j,i,export[j])
  sheet1.write(0,i+1,'stop')
  book1.save('data.xls')
```

5.4. Testovací příklad

Při volbě testovacího příkladu bylo důležité, aby byl dostatečně velký, aby program potřeboval k vyřešení netriviální počet iterací, ale zase dostatečně malý, abych byl schopen odhadnou, jestli program dělá to, co má. Použil jsem síť s 12 vrcholy a 23 hranami, která nemá na počátku zapnutou žádnou hranu. Musím tedy přidat ještě 8 umělých hran. Data k této úloze jsou v přiloženém souboru MS Excel. Na obrázku 5.1 je tento graf znázorněný.



Obrázek 5.1: Graf sítě testovacího příkladu s tučně vyznačenými umělými hranami a tečkovaně vyznačenými nezapnutými hranami.

5.4.1. Testování programu

Při testování programu na uvedené síti jsem nejdříve různě měnil parametry cen, abych sledoval základní chování programu. Vždy jsem větším množstvím spuštění programu našel pravděpodobné globální minimum a následně jsem sledoval potřebný počet iterací k dosažení tohoto optima.
Při těchto prvních testech jsem zjistil, že program má výrazný problém s uváznutím v lokálním. Po určitém počtu kroků se začaly generace skládat ze samých stejných jedinců. Důvodem bylo, že původní použitá metoda tvoření nových generací (viz. 5.3.4) umožňovala přesunutí nejlepších jedinců do nové generace bez mechanismu, který by je odstranil. Křížení dvou stejných jedinců přidá dalšího stejného jedince a redukce již redukovaného jedince přidá další jeho kopii. Jediná operace, která v tu chvíli mohla najít nového lepšího jedince byla mutace, která ale vycházela vždy znovu ze stejného jedince, takže by pravděpodobnost mutace musela být velmi vysoká, aby byla rozumná šance lokální minimum opustit.
Změnil jsem proto postup tvorby nových generací na elitní doplnění, snížil počet potomků generovaných operátorem redukce a velmi zvýšil pravděpodobnost mutace. Situaci to zlepšilo, ale problém s uváznutím v podstatné míře přetrvával. Ukázalo se, že efekt různých operátorů závisí významně na konkrétním zadání. Zvláště operátor mutace má různý efekt podle vzdálenosti lokálního minima od globálního. Pokud lokální minimum, ve kterém program uvázl je vzdálené od globálního, tedy chromozomy těchto dvou řešení se liší v větším množství genů, tak je vhodné zvolit vysokou pravděpodobnost mutace. Opačně to platí pro lokální minimum blízké globálnímu. Časově nejlepší alternativou pro nalezení globálního optima se zdá několikanásobné spuštění algoritmu s předem daným menším počtem iterací a následné vybrání nejlepšího výsledku. Toto ale neplatí pro průměrnou hodnotu výsledku, pro ni je statisticky lepší spouštět algoritmus jednou.

Nakonec jsem provedl testy na datech uvedených v přiloženém souboru. Jako startovací kombinaci parametrů jsem si zvolil

$$n\_populace = 20$$
$$n\_potomků = 20$$
$$n\_krizencu = 18$$
$$mutace = 0.05$$
$$max\_iter = 20$$
$$elita = 3$$

konvergence a *stop* nepoužiji, proto je nastavím nesplnitelně. Popis parametrů viz 5.3.5. Údaje měřím jako průměr ze 100 spuštění algoritmu. Nejlepší dosažená hodnota byla 1705. Tato hodnota je pravděpodobným globálním minimem.
Nejdříve měním hodnotu *mutace*.

Tabulka 5.1: Tabulka změny průměrného výsledku v závislosti na hodnotě *mutace*.

mutace	0.05	0.1	0.2	0.4	0.6
průměrná hodnota výsledku	1720	1719.05	1721.4	1736.55	1738.95

Poté měním počet kříženců mezi potomky, ted hodnotu *n_krizencu*.

Tabulka 5.2: Tabulka změny průměrného výsledku v závislosti na hodnotě *n_krizencu*

n_krizencu	18	20	16	14	10
průměrná hodnota výsledku	1720	1743	1719	1721.4	1727.1

Další parametr co měním je *elita*, tedy počet nejlepších jedinců, co přechází ze staré do nové generace.

Tabulka 5.3: Tabulka změny průměrného výsledku v závislosti na hodnotě *elita*

elita	1	2	3	4	5	7	10	15
průměrná hodnota výsledku	1720.85	1718.5	1720	1719.95	1721.8	1725.25	1725.05	1726.5

Další parametry měnit nebudu, protože ty by měnily přímo výpočetní sílu a výsledky by byly neporovnatelné. Do teď by všechna nastavení měla mít stejnou dobu výpočtu. Průměrná doba jednoho spuštění genetického algoritmu je 0.394 s. Z naměřených hodnot by měla být nejlepší kombinace

$$n\_krizencu = 16$$
$$mutace = 0.1$$
$$elita = 2$$

Když algoritmus spustím s těmito parametry, dostávám výsledek 1717.25, což je zatím nejlepší průměrný výsledek. Takto mohu pokračovat dále a s jemnějšími změnami parametrů, ale brzo narazím na rozptyl průměrných výsledků. V tu chvíli bych musel zvýšit počet spouštění algoritmu, což ale vede k zvýšení časové náročnosti. Při počtu spuštění 100 byla celková doba zkoušení průměrně 39,4 s.

6. Závěr

V práci jsou shrnuty podklady k pochopení problému toku sítí se zapínáním hran. Dále popisuje strukturu genetických algoritmů. Tyto informace jsou aplikovány při tvorbě konkrétního algoritmu. Jeho principem je rozdělit problém na dvě části, kdy jednu z nich řeší síťová simplexová metoda volaná příkazem a druhou genetický algoritmus naprogramovaný v programovacím jazyce Python. Pro daný problém jsou vymyšlena data k testování algoritmu. Po testování je algoritmus upraven a vhodně nastaven. V práci je příklad postupu takového nastavování. Algoritmus je připraven na import dalších dat.

Literatura

[1] BITARA, Matúš Síťové úlohy a jejich modifikace: bakalářská práce. BRNO: Vysoké učení technické v Brně, Fakulta strojního inženýrství, Ústav Matematiky, 2016. 47 s. Vedoucí práce byl RNDr. Pavel Popela, Ph.D

[2] DEMEL, Jiří. Grafy a jejich aplikace. Praha: Academia, 2002. ISBN 80-200-0990-6.

[3] HAUPT, Randy L. a S. E. HAUPT. Practical genetic algorithms. 2nd ed. Hoboken, N.J.: John Wiley, c2004. ISBN 0-471-45565-2.

[4] HRABEC, D.Mathematical Programs for Dynamic Pricing - Demand Based Management. Brno: Vysoké ucení technické v Brne, Fakulta strojního inženýrství, 2016. 119 s. Vedoucí disertacní práce prof. Kjetil Kåre Haugen, PhD

[5] HYNEK, Josef. Genetické algoritmy a genetické programování. Praha: Grada, 2008. Průvodce (Grada). ISBN 978-80-247-2695-3.

[6] KLAPKA, Jindřich, Pavel POPELA a Jiří DVOŘÁK. Metody operačního výzkumu. Vyd. 2. Brno: VUTIUM, 2001. ISBN 80-214-1839-7

[7] KOMÍNEK, Jan Heuristické algoritmy pro optimalizaci: diplomová práce. Brno: Vysoké učení technické v Brně, Fakulta strojního inženýrství, Ústav matematiky, 2012. 77 s. Vedoucí práce byl Ing. Jan Roupec, Ph.D.

[8] NASH, Stephen a Ariela. SOFER. Linear and nonlinear programming. New York: McGraw-Hill, c1996. ISBN 978-0070460652.

[9] Python Software Foundation. Python Language Reference, 3.6. Available at http://www.python.org

[10] REEVES, Colin R. a Jonathan E. ROWE. Genetic algorithms: principles and perspectives : a guide to GA theory. Boston: Kluwer Academic Publishers, c2003. ISBN 1-4020-7240-6.

7. Seznam příloh

1. zapinani_uzlu.py - soubor s hybridním algoritmem
2. data.xls - soubor MS Excel, z kterého hybridní algoritmus importuje data, a do kterého data exportuje